OrientX 系统程序员开发手册

版本号：OrientX Version 2.0
完成时间：2004 年 9 月 10 日
开发单位：中国人民大学 IDKE 实验室 XML 工作组

1. 引言

1.1 编写目的

本说明书主要是介绍如何基于 OrientX 系统进行应用开发，包括简单介绍 OrientX 系统的主要功能、示例说明、各个模块的应用接口等。

1.2 系统概述

随着因特网应用的发展，XML 逐渐成为数据描述和数据交换的标准，大量的 XML 文档出现在网络中；有效地存储 XML 数据并提供高效的 XML 数据查询，成为当前急需解决的问题。

OrientX 系统正是在这样的应用背景下产生的，它以 Native 方式存储 XML 数据，并支持 XPath 和 XQuery 等 XML 查询以读取数据。所谓的 XML 的 Native 存储方式，就是存储时保留数据的树形模式；根据一个结点可以直接找到其孩子结点、左右兄弟结点或父亲节点等。以 Native 方式存取 XML 数据，就无需进行数据模式的转换，也不需要进行查询语言的转换。

OrientX 是一个 Native XML 数据库管理系统（Native XML DataBase Management System）。OrientX 是 **O**riginal **R**UC **I**DK**E** **N**a**t**ive **X**ML 的缩写，表示 OrientX 系统是中国人民大学 IDKE 实验室研发的 Native XML 数据管理系统。该系统的研发工作是从 2001 年至今，已经历时 3 年时间。

若想更详细地了解 OrientX 系统，请参照文档“OrientX 系统说明书”。

1.2.1 系统功能

OrientX 是 Native XML 数据库管理系统，主要功能如下的：

1. 数据库的建立和维护
2. 数据操纵,包括数据导入/导出，数据检索，数据更新等。
3. 数据库运行管理
4. 数据组织、存储和管理功能
5. 数据通信接口

1.2.2 系统特征

OrientX 系统作为一个数据库管理系统，其系统特征具有如下的

1. 它是基于模式（Schema based）的 Native XML 数据库管理系统。
2. 多样化的数据组织和存储方式
3. 数据存取路径
4. 数据模型遵循 XQuery 1.0 and XPath 2.0 Data Model 标准。
5. 数据检索支持 W3C 推荐的 XQuery1.0 标准

1.3 如何得到 OrientX 系统

OrientX 系统是中国人民大学 IDKE 实验室研发的成果。版权和所有权归中国人民大学 IDKE 实验室所有。但其内容可以免费提供予任意与商业利益无关的实验和研究活动。

如果用户需要 OrientX 系统相关的程序包、源代码、说明文档，请向我们索取：OrientX@ruc.edu.cn 或登录我们的主页：http://idke.ruc.edu.cn/OrientX

关于 OrientX 系统的功能演示，请登录我们的主页：http://idke.ruc.edu.cn/OrientX

1.4 OrientX Mailing List

如果您在使用 OrientX 系统过程中，遇到问题，请参照 OrientX 系统的相关文档；或者登录我们的主页：http://idke.ruc.edu.cn/OrientX 参考常见问题解答(Frequent Questions)；或者给我们发 email：OrientX@ruc.edu.cn ，我们将尽快给您回复解答。若您对我们的系统工作有什么意见，欢迎反馈：OrientX@ruc.edu.cn 。

2.系统结构

描述 OrientX 系统的总体架构的最好方法是，通过检查它是如何导入一个 XML 文档并存储到数据库中去；然后在其上进行一个 XML 查询，看看它是如何实现的，如何导出文档的实现过程。

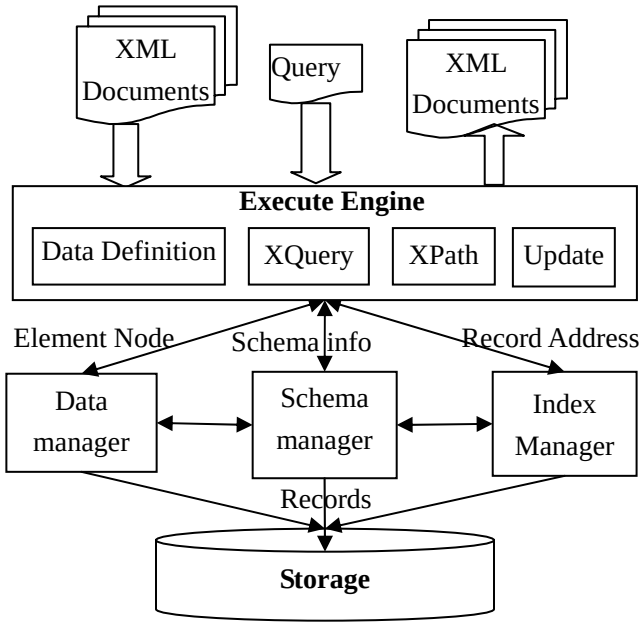


图 1　**系统框架图**

如图 1 所示，用户用 OrientX 系统管理 XML 数据，首先需要通过执行引擎（Execute Engine）模块建立一个数据库（前面已经说过，OrientX 系统中每一个数据库对应于具有相同的模式的数据集 dataset）。这就是数据定义（Data Definition），它确定了数据集内的所有文档的模式结构。导入文档时，执行引擎把文档传送到数据管理（Data Manager）模块；数据管理模块从逻辑上把 XML 文档划分成多个记录。然后传输到存储（Storage）模块，选择

适当的文件机构进行存储。

当需要对数据进行查询检索时，一个 XML 查询（XPath 或者 XQuery 查询）以文本的形式传送到查询执行引擎（XQuery 和 XPath 执行引擎的处理处理策略有很大不同）；在查询执行引擎中，XML 查询将被分析(parse)成一个查询执行计划，此过程中从模式管理模块（Schema manager）读取相关信息，判断该查询是否存在语义错误，例如目标文档或数据库是否存在，XPath 路径中的结点在对应的模式中是否存在等问题；如果存在这样的错误，则系统就报告错误，查询不再往下执行。查询执行引擎还可以对查询计划进行优化；如果存在合适的索引可以优化查询执行效率，查询执行引擎就可以通过索引管理模块（index manager）直接访问数据库，而不需要通过数据管理模块（Data Manager）导航式地访问数据库。

若想更详细地了解 OrientX 系统各个模块的设计，请参照文档“OrientX 系统说明书”以及各个模块的概要设计说明书。

3.运行环境

3.1 配置要求：

OrientX 系统运行的硬件平台要求：

- 操作系统：Windows 98/NT/2K/XP
- 编译器：Microsoft Visual C++ 6.0
- 最低的可用空间要求：10MB
- 最低的内存空间要求：64MB

3.2 目录设置(Dictionaries Settings)：

1. 以下的操作步骤假定您的工程所在的目录为 D:\OrientXAPI，如果您的工程文件在另外的位置，请对目录做相应调整。
2. 在 VC++中选择“Tools”菜单下“Options”菜单项，选择“Directories”面板，在“Show directories for”下拉框中选中“Include files”选项，添加一个目录 “D:\OrientXAPI\OrientX”。
3. 以上各步完成后，单击“OK”按钮，确认修改。

3.3 添加库文件(Add Lib Files to Project)：

1. 在 VC++中选择“Project”菜单下“Add to Project”，“Files”菜单项，弹出“Insert Files into Project”对话框，在“查找范围”下拉框中选择目录“D:\OrientXAPI”，在“文件类型”下拉框中选择“所有文件 (*.*)”，选中“OrientXd.lib”，单击“OK”按钮确认。

3.4 工程设置(Project Settings)：

OrientX 系统 2.0 版本只支持在 Debug 环境下进行开发，其工程设置如下：
1. 在 VC++中选择“Project”菜单下“Settings”菜单项，选择“Link”面板，在“Ignore libraries”文本框中添加一个库文件“libcd.lib”。

2. 在 VC++中选择“Project”菜单下“Settings”菜单项，选择“C/C++”面板，在“Code Generation”下拉框中选中“Code Generation”选项，在“Use run-time library”下拉框中选中“Debug Multithreaded”选项。
3. 以上各步完成后，单击“OK”按钮，确认修改。

4 使用示例

OrientX 2.0 版本中把系统所支持的主要功能封装到 CExecuteEngine 类中，向最终用户提供简单统一的接口，实现建库/删库，导入/导出/删除文档，查询，更新等功能。

4.1 系统初始化和结束

系统的初始化和系统结束时的资源释放，分别通过调用函数 CExecuteEngine::Initialize()和 CExecuteEngine::clear()实现。下面的程序演示了系统的主要功能；程序保存在文档“OrientX.cpp”中。

示例程序：

```
#include "include/share/ExecuteEngine.h"
int main()
{
    //to initialize the system.
    CExecuteEngine::Initialize();

    //create a database name "bib"
    CExecuteEngine::CreateDataSet("bib", "E:\\OrientX_2_0_API\\OrientX\\example\\bib.xsd");
    CExecuteEngine::PrintDataSetName();
    cout<<endl;

    //to import a document to database "bib"
    CExecuteEngine::ImportDoc("bib", " E:\\OrientX_2_0_API\\OrientX\\example\\bib.xml");

    //to export the document "bib.xml" in the database "bib"
    CExecuteEngine::ExportDoc("bib", "bib.xml", "output_bib.xml");
    CExecuteEngine::PrintDocName("bib");
    cout<<endl;

    //to commit an XQuery with the query kept in file "Xquery.txt"
    CExecuteEngine::ExecXQuery("E:\\OrientX_2_0_API\\OrientX\\example\\XQuery.txt",
"XQueryResult1.xml") ;

    //to commit an XQuery with the query in the parameter directly. note the 3rd parameter.
 CExecuteEngine::ExecXQuery("import default schema \"bib\"    document('bib@bib.xml')
 //book[price > 100]", "XQueryResult2.xml", false) ;
```

```
    //to commit an XPath    with the query in the parameter directly. note the 3rd parameter.
    CExecuteEngine::ExecXPath("document(\"bib@bib.xml\")//author", "XPathResult.xml",
false);

    //to commit an update in file.
    CExecuteEngine::ExecXQuery("E:\\OrientX_2_0_API\\OrientX\\example\\update.txt");
    CExecuteEngine::ExportDoc("bib","bib.xml","bib_update.xml");

    //to drop the document and dataset.
    CExecuteEngine::DropDoc("bib","bib.xml");
    CExecuteEngine::DropDataSet("bib");
    CExecuteEngine::clear();

    return 0;
}
```

通过 CExecuteEngine 只能提交任务，任务的运行结果（如果有的话）被输出到指定的文件中。但是无法直接访问到其内部的数据，例如用户通过 CExecuteEngine 提交了一个 XQuery，其结果输出到了指定的文件中，用户无法对查询的结果作进一步的开发！因此需要直接调用各个模块的接口。下面就是实现各个模块的功能示例程序。

4.2 查询执行（Query Process）

4.2.1XML 查询分析

下例示范如何声明 XPath 查询分析器，如何输入 XPath，并对其进行语法分析，得到相应的执行计划，然后再解析成 XPath 执行引擎所能够识别的内部结构。存放下列代码的文档名为“XPathParser.cpp”

```
#include "include/QueryParser/Xpath_parser.h"
int main()
{
    //初始化系统
    //… ….
    //XPath查询分析器
    Xpath_parser parser;
    //重置输入查询的文档路径和文档名
    parser.SetInputFile("c:\\input.txt");
    //重置输出分析过程的文档路径和文档名
    parser.SetOutputFile("c:\\DebugOutput.txt");
    //重置输出错误信息的文档路径和文档名
    parser.SetErrorFile("c:\\ErrorMessage.txt");
```

```
    //开始分析
    if( parser.Run() == 0)
        return -1;//分析失败：输入的查询有误

    //读取分析结果：用内部结构表示的XPath查询
    CxpXpathExpr* XPath=parser.GetXpath();

    //后续操作，例如传递给XPath查询执行引擎，释放系统资源等
    //... ...

}
```

4.2.2 XPath 查询执行

下面是 XPath 查询引擎的使用示例。说明如何对经过语法分析的 XPath 语句进行查询优化和执行。

```
#include "include/publicHeaders/stdafx.h"
#include "include/Datamanager/NxdbDataManager.h"
#include "include/queryprocess/QueryProcess.h"
#include "include/QueryParser/Xpath_parser.h"
#include "include/share/ExecuteEngine.h"

//引用外部变量
extern NxdbDataManager g_DataManager;
extern Xpath_parser g_XPathParser;
extern QueryProcess g_XPathExecute;

int main()
{
    //前续操作，初始化系统，建库导入文档，分析 XPath 查询等
    //… …

    //读取分析结果：用内部结构表示的 XPath 查询
    CxpXpathExpr* XPath= g_XPathParser.GetXpath();

    //假定输入的查询是一个完整的 XPath 查询，即以 document("XMLFile.xml")开始
    NxdbList result;
    g_XPathExecute.EvaluateXPath(NULL,XPath,result);

    //把结果集 result 打印到指定的文件 resultFile 上
    outputResult("C:\\OrientX\\test\\result.xml", result);

}
```

说明：在执行这段程序之前，需要将XPath语句写入分析器的输入文件（inputfile.txt）；执行的结果输出到文档“result.xml”，同时保存在变量result中，以待下一步处理。

4.2.3 XQuery 导航式处理

下面的程序演示了如何使用XQuery执行引擎进行数据检索，还可以在其执行结果上进行进一步的开发。下面的程序存放在文件“XQueryTest.cpp”中。

```
#include "INCLUDE/PublicHeaders/XQExeNavigation.h"
#include "include/share/ExecuteEngine.h"
extern CxqeXQueryExecute g_XQueryExecute;

int main()
{
    CExecuteEngine::Initialize();
    g_XQueryExecute.initialize();

    char* inputfile = "E:\\develop\\OrientX_2_0_API\\OrientX\\example\\XQuery\\";
    char* outputFile = "E:\\develop\\OrientX_2_0_API\\OrientX\\example\\XQueryResult.txt";

    //to commit an XQuery and evaluate it.
    if( g_XQueryExecute.run(inputfile,outputFile) == -1)
    {
        cout<<"failed."<<endl;
        return -1;
    }
    else
        cout<<"OK."<<endl;

    const CComputeEleNode* ele = g_XQueryExecute.getResultData();

    //...
    FILE* fp = fopen("tmpFile.xml","w");
    ele->print2File(fp);

    //to develop application based on the OrientX's XQuery engine.

    g_XQueryExecute.clear();
    CExecuteEngine::clear();
    return 0;
}
```

4.3 创建/删除数据库，导入/导出文档

下面的示例程序介绍了如何建立、删除数据集，如何导入、导出和删除文件。存放下列代码的文档名为“datamanagementTest.cpp”

```
//要包含的头文件
#include "include/share/ExecuteEngine.h"
#include "include/datamanager/nxdbdatamanager.h"

//引用外部变量
extern    NxdbDataManager g_DataManager;

int main()
{
      //initialize the system

      //create a new dataSet
      if( g_DataManager.CreateDataSet("Xmark"," xmark.xsd") == -1)
            return -1;

      //import a new document
      if( g_DataManager.ImportDoc("Xmark"," xmark1.xml") == -1)
            return -1;

      //export the specified document with the xml format
      if( g_DataManager.ExportDoc("xmark","xmark1.xml"," output1.xml") == -1)
            return -1;

      //delete the specified document
      if( g_DataManager.DropDoc("xmark","xmark1.xml") == -1)
            return -1;

      //delete the dataset
      if( g_DataManager.DropDataSet("xmark") == -1)
            return -1;

      //clear the system and release resource
}
```

说明：

如果需要重新设定存放数据的目录，则在初始化系统之后，立刻调用函数SetDataPath(). 例如：设置Data目录为C:\OrientX\Data，则在函数g_SysManager.initiate()之后，调用函数 g_DataManager.SetDataPath("c:\\orientx")。

4.4 创建/删除索引

下面的示例演示了如何建立路径索引，打开索引，并利用路径索引进行结构连接(structure join)，最后关闭索引。此程序代码存为文件 indextest.cpp。

```
#include "include/share/ExecuteEngine.h"
#include "include/accessmanager/nxdbsm.h"
#include "include/indexmanager/pathindex/PIndexManager.h"

extern NX_AccessManager g_AccessManager;
extern PIndexManager g_PathIndexManager;

int main()
{
	CExecuteEngine::Initialize();

	CExecuteEngine::CreateDataSet("xmark",
"E:\\project\\Orientx2.0Server\\OrientX_2_0\\data\\books.xsd");
	CExecuteEngine::PrintDataSetName();
	cout<<endl;

	CExecuteEngine::ImportDoc("xmark",
"E:\\project\\Orientx2.0Server\\OrientX_2_0\\data\\books.xml");


	//here suppose dataset "XMark" already exits in database.
	int dataset = g_AccessManager.OpenDataSet("xmark");
	g_PathIndexManager.CreatePathIndex(dataset);
	NX_DocList doclist = g_AccessManager.GetDocList(dataset);

	for (int i=0; i< doclist.getLength(); i++){
		char* doc = doclist.getName(i);
		int docId = g_AccessManager.OpenDoc(dataset, doc);
		g_PathIndexManager.InsertDoc2PathIndex(dataset, docId);
	}

	//get the two DTDNodeCodes for join(section and title are two nodes defined in books.xsd)
	DTDTree* dtdTree = MetaDataManager::DTDLoader(dataset);
	DTDNodeListItem* itemlist1 = dtdTree->GetDTDNodeList("section");
	DTDNodeListItem* itemlist2 = dtdTree->GetDTDNodeList("title");
	DTDNodeCode code1 = itemlist1->node->getEid();
	DTDNodeCode code2 = itemlist2->node->getEid();

	//we do not need to delete the two DTDNodeListItem as they will be callbacked in the routine
below
```

```
    MetaDataManager::DTDFlusher(dtdTree);


    g_PathIndexManager.OpenPathIndex(dataset);
    //join processing
    SearchResult* result = g_PathIndexManager.AncestorDescendant(code1, code2);
    if (result)
        printf("section and title's join result size is %d\n",result->GetResultSize());
    delete result;
    g_PathIndexManager.ClosePathIndex();

    //delete path index
    g_PathIndexManager.DeletePathIndex(dataset);

    CExecuteEngine::DropDataSet("xmark");

    CExecuteEngine::clear();

    return 0;
}
```

4.5 更新

更新模块借用的是 XQuery 模块的查询引擎，可以说是 XQuery 上的一个开发实例，下面的示例程序演示了更新模块的使用，程序放在文件”XupdateTest.cpp”中，该程序和文件“XQueryTest.cpp”中的示例程序很类似。用户可以用更新语言实现自己的更新。所以第 5.6 节还将介绍从 XQuery 扩展的的更新语言。

文件 u8.up 的更新语句：

```
import default schema     "bib"
for $b in document("bib@bib.xml")/book[title = "TCP/IP Illustrated"]

insert   after   $b   element   values
   <book >
      { $b/author }
      <title>"Thinking in Java"</title>
   </book>
```

```
#include "INCLUDE/PublicHeaders/XQExeNavigation.h"
#include "include/share/ExecuteEngine.h"
extern CxqeXQueryExecute g_XQueryExecute;

int main()
{
```

```
    CExecuteEngine::Initialize();
    g_XQueryExecute.initialize();

    char* inputfile = "E:\\develop\\OrientX_2_0_API\\OrientX\\example\\Update\\u8.cpp ";
    char* outputFile = "E:\\develop\\OrientX_2_0_API\\OrientX\\example\\XQueryResult.txt";

    //to commit an XQuery and evaluate it.
    if( g_XQueryExecute.run(inputfile,outputFile) == -1)
    {
        cout<<"failed."<<endl;
        return -1;
    }
    else
        cout<<"OK."<<endl;

    //...

    //to develop application based on the OrientX's XQuery engine.

    g_XQueryExecute.clear();
    CExecuteEngine::clear();
    return 0;
}
```

该更新语句的含义是：在文档 bib.xml 的某个 book 元素后插入新构造的元素，该 book 的 title 的值是"TCP/IP Illustrated"，新构造的元素的 title 是"Thinking in Java"，作者和前一个 book 的作者一样。

5.各模块的 API 接口

5.1 存取管理(Access Manager)

存取模块的功能通过NX_AccessManager类来提供。OrientX系统中定义了NX_AccessManager类型的全局变量 g_AccessManager。当用户需要调用存取模块的功能只需要包含头文件#include "include/accessmanager/nxdbsm.h",并声明g_AccessManager为外部变量即可使用它。

NX_AccessManager 类提供的函数接口如下：

1. 创建数据集

int CreateDataSet(char* DataSetName);

区分本函数和 DataManager 中的同名函数，后者实现的时候会调用本函数，本函数生成 FreePageXXXXX.ctl 和 SetXXXXX.ctl 两个文件，分别负责新生成的数据集的存储空间的管理和数据文件的管理。DataManager 中的同名函数除了调用这个函数外，还会生成相关的 oid 集等。

参数说明：

char* DataSetName：指定新的数据集的名称

返回值：成功返回 0，否则返回相应的错误代码。

2. 删除数据集

int DeleteDataSet(char* DataSetName);

本函数删除该数据集所有数据文档对应的物理文件，删除该数据集相关的 FreePageXXXXX.ctl 和 SetXXXXX.ctl 文件。会被 DataManager 中的同名函数调用。

参数说明：

char* DataSetName：指定要删除的数据集的名称

返回值：成功返回 0，否则返回相应的错误代码。

3. 在指定数据集中建立数据文档文件

int CreateDoc(char* DataSetName,char* DocName,int Type);

参数说明：

char* DataSetName：要生成的数据文件所属的数据集

char* DocName：要生成的数据文件的名称

int Type：数据文件的类型

返回值：成功返回 0，否则返回相应的错误代码。

4. 在指定数据集中删除指定的文档

int DeleteDoc(char* DataSetName,char* DocName);

参数说明：

char* DataSetName：要删除的文件所属的数据集

char* DocName：要删除的文件名称

返回值：成功返回 0，否则返回相应的错误代码。

5. 保存文件的根地址信息

int WriteDocRoot(char* DataSetName,char* DocName,int lpno,int RecordNo);

参数说明：

char* DataSetName：要保存的文档所属的数据集的名称

char* DocName：要保存的文档的名称

int lpno：要保存的文档的根节点的地址（逻辑页号）

int RecordNo：要保存的文档的根节点的地址（页内记录号）

返回值：成功返回 0，否则返回相应的错误代码。

6. 读取文件的根地址信息

int ReadDocRoot(char* DataSetName,char* DocName,int& SetId,int& DocId,int& lpno,int& RecordNo);

参数说明：

char* DataSetName：要读取得文件所属的数据集的名称

char* DocName：要读取根结点信息的数据文件的名称

int& SetId：该数据集对应的 id

int& DocId：该数据文件对于的 id

int& lpno：根结点所在的页号

int& RecordNo：根结点的页内记录号

返回值：成功返回 0，否则返回相应的错误代码。

7. 在指定数据集中寻找新的空页面

int FindEmptyPage(int SetId);

参数说明：

int SetId：要寻找数据集的 id

返回值：成功返回 0，否则返回相应的错误代码。

5.2 数据管理(Data Manager)

数据管理模块的主要功能是建立数据集、删除数据集，导入数据、导出数据，向上层查询模块提供数据访问。通过 NxdbDataManager 类管理数据集的建立和删除，数据文档的导入导出和删除，通过 ElementNode 等类实现向上层提供元素结点间的导航。

相应的头文件：

“include/datamanager/nxdbdatamanager.h”

“include/publicheaders/storageheads.h”

相应的变量的声明；NxdbDataManager 定义的全局变量 g_DataManager，在需要的地方用 extern 语句引用 g_DataManager 为外部变量。

5.2.1 NxdbDataManager 类提供的函数接口

1. 指定目录DATA的存放路径

功能：Data目录是用来存放OrientX系统中数据文档的具体内容和相关的模式信息等。本函数指定Data目录要存放在哪个路径下，如果该目录存放在OrientX文件夹下，则不用调用该函数，或者以参数NULL调用。

int setDataDirectory(char* pathName = NULL);

参数说明：

char* pathName：非空值给出新的存放Data目录的路径

返回值：成功返回0，否则返回-1。

2. 建立数据集

功能：根据指定的数据集模式建立相应的数据集，包括为它分配一定的存储空间，用来存储该数据集的文档或索引，建立相应的dtdtree或者blocktree，生成相应的oid集等。

int CreateDataSet(char* dataSetName,char* dtdFileName);

参数说明：

char* dataSetName：要建立的数据集的名称

char* dtdFileName：和该数据集相关的模式文档的名称

返回值：成功返回0，否则返回-1。

3. 删除数据集

功能：从 OrientX 系统中删除指定名称的数据集，收回分配给它的存储空间，同时会收回相应的 oid 集的存储空间。会删除该数据集内所有的文档。

int DropDataSet(char* dataSetName);

参数说明：

char* dataSetName：要删除的数据库的名称

返回值：成功返回 0，否则返回-1。

4. 导入文档

功能：导入文档到指定的数据集中

int ImportDoc(char* dataSetName,char* docName ,int Type = NX_XML_DATA_TYPE,StorageMode storageMode = DEBMode);

参数说明：

char* dataSetName：要将文档导入到的数据集的名称，该数据集的模式应该和该文档的模式相同，否则会报错。

char* docName：要导入的文档的名称

int Type ：说明要导入的是否为索引文件，缺省值为否

StorageMode storageMode：指定要导入的文档的存储模式，缺省值为 DEBMode

返回值：成功返回 0，否则返回-1。

5. 导出文档

功能：导出指定的文档为 XML 格式，并指定导出后的文档的名称

int ExportDoc(char* dataSetName,char* oldDocName,char* newDocName);

参数说明：

char* dataSetName：要导出的文档所属的数据集的名称

char* oldDocName：要导出的文档的名称

char* newDocName：指定要导出的文档的新名称

返回值：成功返回 0，否则返回-1。

6. 删除文档

功能：删除指定的文档

int DropDoc(char* dataSetName,char* docName);

参数说明：

char* dataSetName：要删除的文档所属的数据集的名称

char* docName：要删除的文档的名称

返回值：成功返回 0，否则返回-1。

7. 结果输出函数

功能：输出 xpath 查询的结果，是一些 element node 的列表，目前 xpath 的查询都是针对单个数据集进行的，并且都只是从原文档中抽取出来的数据，没有自己构造的数据，也没有改变它们间的结构信息。

char* ResultExport(char* dataSetName,ElementNodeList* nodeList);

参数说明：
char* dataSetName：指名要输出的数据属于哪个数据集
ElementNodeList* nodeList：要输出的结点的列表
返回值：成功返回 0，否则返回-1。

8. 数据导航

函数功能：读取指定文档的根元素
ElementNode* GetRootElement(char* dataSetName,char* docName);
参数说明：
char* dataSetName,指定的数据集的名称
char* docName 指定的文档的名称
返回值：如果成功则返回根元素的指针，否则返回 NULL

9. 数据导航

函数功能：读取指定文档的根元素，和上一个函数的功能相同。只是一个参数是名称，另一个函数的参数是 id。
ElementNode* GetRootElement(int setID, int docID);
参数说明：
int setID, 指定的数据集的 id
int docID 指定的文档的 id
返回值：如果成功则返回根元素的指针，否则返回 NULL

5.2.2 ElementNode 类提供的函数接口

1. 取得当前 element 结点的父结点

ElementNode* GetParent(short& flag);
参数说明：
short& flag：保存可能的错误信息
返回值：如果成功则返回找到的父结点的指针，否则返回 NULL。

2. 取得当前 element 结点的指定 eid 的左兄弟结点

ElementNode* GetLeftSibling(DTDNodeCode eid,short& flag);
参数说明：
DTDNodeCode eid：指定要寻找的左兄弟结点的 eid
short& flag：返回可能的错误代码
返回值：如果找到则返回找到的结点的指针，否则返回 NULL。

3. 取得当前 element 结点的指定 eid 的右兄弟结点

ElementNode* GetRightSibling(DTDNodeCode eid,short& flag);
参数说明：
DTDNodeCode eid：指定要寻找的左兄弟结点的 eid
short& flag：返回可能的错误代码

返回值：如果找到则返回找到的结点的指针，否则返回 NULL。

4. 取得当前 element 结点的指定 eid 的最左边的子结点

ElementNode* GetFirstChild(DTDNodeCode eid,short& flag);

参数说明：

DTDNodeCode eid：指定要寻找的子结点的 eid

short& flag：返回可能的错误代码

返回值：如果找到则返回找到的结点的指针，否则返回 NULL。

5. 取得当前 element 结点的指定 eid 的最右边的子结点

ElementNode* GetLastChild(DTDNodeCode eid,short& flag);

参数说明：

DTDNodeCode eid：指定要寻找的子结点的 eid

short& flag：返回可能的错误代码

返回值：如果找到则返回找到的结点的指针，否则返回 NULL。

6. 取得当前 element 结点的指定 eid 的所有后代结点

NxdbObjectList* GetDecendent(DTDNodeCode eid,short& flag);

参数说明：

DTDNodeCode eid：指定要寻找的后代结点的 eid

short& flag：返回可能的错误代码

返回值：如果找到则返回找到的结点的指针，否则返回 NULL。

7. 在当前 element 下增加属性

short AppendAttrAfter(Attr* a, Attr* insertPoint);

参数说明：

Attr* a：要增加的属性的指针

Attr* insertPoint：指定要增加到哪个属性后面

返回值：成功返回 0，否则返回-1。

8. 取得当前 element 的指定属性的值

void* GetAttrValue(DTDNodeCode& aid,DataType& aType,unsigned int& aLen);

参数说明：

DTDNodeCode& aid：指定要取值的属性的 aid

DataType& aType：返回要取值的属性的类型

unsigned int& aLen：返回要取值的属性的长度

9. 取当前 element 的值内容

void* GetTextValue(DataType& tType,unsigned int& tLen);

参数说明：

DataType& tType：返回元素内容的值类型

unsigned int& tLen：返回元素内容的长度

5.3 模式管理(Schema Manager)

Schema 类似与关系数据库中的数据字典。数据字典说明了数据库中的表结构、索引结构等信息。在 OrientX 中，一个数据集对应于一个 Schema，每一个数据集由多个符合同样 Shema 定义的文档组成。Schema 有两大作用：首先，Schema 类似于 DTD 或者 XMLSchema 的作用，说明了文档的结构特征，结点的数据类型。其次，Schema 还记录了关于数据集的索引、引用和被引用关系等。

SchemaManager 有五个数据结构类（Attribute, DTDNode、DTDTree，BlockNode，BlockTree）和一个管理类（MetaDataManager）。

MetaDataManager 类负责 DTDTree 和 BlockTree 的保存（到磁盘）和（从磁盘）读取。

Attribute 类和 DTDNode 类分别对应于 DTD 中的 Attribute 结点和 DTD 结点。DTDTree 则代表由 DTDNode 和 Attribute 组成的整个 Schema。DTDTree 提供由 TagName 到 ID 的双向映射方法。

DTDNode 类代表 DTDTree 上的 DTD 结点。ElementNode 结点是 DTDNode 结点的实例。DTDNode 类说明了属性的名字，ID，数据类型，是否有索引，是否有递归环等。

DTDTree 类是 Schema Tree 的抽象。它以一个 DTDNode 为根。DTDTree 上记录了该 Schema 的实例文档数目、占用的物理块的数目等信息。同时，DTDTree 还提供从 TagName 到 EID 和 EID 到 TagName 的双向映射。在导入导出文档和进行查询时，DTDTree 作为数据集的 Schema，将会被频繁使用。

BlockNode 和 BlockTree 是实现聚簇类存储方法(CSB, CEB)方法时需要的一个数据结构。聚簇类存储方法把 DTDTree 分解成若干语义块(Block)，根据语义块把文档分解成实例块(instance block)。符合某一个语义块的所有实例块聚簇存储。BlockNode 表示一个语义块，记录了符合该语义块的所有实例块的信息，比如这些实例块占用的物理块数等。BlockTree 则是 BlockNode 组成的树。每一个用 CSB 或者 CEB 存储的文档都有一个 BlockTree。

BlockTree 类代表语义块组成的树。它提供了查找 BlockNode，从 DTDTree 生成 BlockTree 等方法。每一个聚簇方法存储的数据集有一个总的 BlockTree，该数据集中的每一个文档有一个 BlockTree。我们把前者叫作 Dataset BlockTree，后者叫做 Document BlockTree。

5.3.1MetaDataManager

1. 导入指定数据集的 DTDTree

static DTDTree* DTDLoader(char* DataSetName);

参数说明：

char* DataSetName：数据集的名称。

OrientX 系统支持有模式约束的 XML 文档，所以每个数据集都有一个模式文档与之相对应，而该模式文档在内存中的表现就是一个 DTDTree，故通过数据集的名称取得与之相关的 DTDTree。

返回值：成功返回模式树的指针，否则返回 NULL。

static DTDTree* DTDLoader();

参数说明：

int setID：OrientX 系统给数据集分配的唯一的 id。

返回值：成功返回模式树的指针，否则返回 NULL。

2. 导入指定数据集的 DataSetBlockTree

static BlockTree* DataSetBlockTreeLoader(char* DataSetName,StorageMode mode);

参数说明：

char* DataSetName：指定数据集的名称。

如果文档存储的模式是聚簇的，需要有个 BlockTree 用来记录聚簇信息。而每个数据集有个数据集相关的 BlockTree，每个文档有个文档相关的 BlockTree。本函数导入的是数据集相关的 BlockTree。

StorageMode mode：指明是 CEBMode 还是 CSBMode。

返回值：成功返回 BlockTree 的指针，否则返回 NULL。

3. 导入指定数据集的 DataSetBlockTree

static BlockTree* DataSetBlockTreeLoader(int setId,StorageMode mode);

参数说明：

int setId：要导入数据集的 id。

StorageMode mode：指明是 CEBMode 还是 CSBMode。

返回值：成功返回 BlockTree 的指针，否则返回 NULL。

5.3.2DTDNode

1. 得到指定 ID 的属性

Attribute* GetAttribute(DTDNodeCode& AID);

参数说明：

DTDNodeCode& AID：指定要读取的属性的编码。

返回值：如果成功返回读取的属性的指针，否则返回 NULL。

Attribute* GetAttribute(char* attName);

参数说明：

char* attName：指定要读取的属性的名称。

返回值：如果成功返回读取的属性的指针，否则返回 NULL。

2. 得到指定 ID 或名字的孩子结点

DTDNode* GetChild(DTDNodeCode& eid);

参数说明：

DTDNodeCode& eid：指定子结点的编码。

返回值：如果成功返回读取的 DTDNode 的指针，否则返回 NULL。

DTDNode* GetChild(char* childName);

参数说明：

char* childName：指定子结点的名称。

返回值：如果成功返回读取的 DTDNode 的指针，否则返回 NULL。

3. 得到第一个指定 ID 的右兄弟结点（不一定是直接右兄弟结点）

DTDNode* GetRightSibling(DTDNodeCode& eid);

参数说明：

DTDNodeCode& eid：指定右兄弟结点的编码。

返回值：如果成功返回读取的 DTDNode 的指针，否则返回 NULL。

4. 得到指定名字的子孙结点的列表

DTDNodeListItem* GetDescendant(const char* name);

参数说明：

const char* name：得到指定名称的所有后代子孙结点的列表。

返回值：如果成功返回读取的 DTDNode 的指针列表，否则返回 NULL。

5. 得到指定编码的后代结点

DTDNode* GetDescendant(DTDNodeCode& nodeID);

参数说明：

DTDNodeCode& nodeID：指定结点的编码

返回值：如果成功返回读取的 DTDNode 的指针列表，否则返回 NULL。

6. 得到按照前序遍历顺序第 index 个子孙结点（如果 index=0，返回本结点）

DTDNode* GetDescendant(int index);

参数说明：

int index：指定子孙结点的序号。

返回值：如果成功返回读取的 DTDNode 的指针列表，否则返回 NULL。

7. 得到指定 ID 的祖先结点。由于有递归环的存在，一个结点可能有若干个父亲结点

DTDNode* GetAncestor(DTDNodeCode parentEID);

参数说明：

DTDNodeCode parentEID：指定的父结点的 eid。

返回值：如果成功返回读取的 DTDNode 的指针列表，否则返回 NULL。

得到第 index 个祖先结点，如果 index=0，返回父亲结点；如果 index=1，返回父亲的父亲结点

DTDNode* GetAncestor(int index);

参数说明：

int index：如果 index=0，返回父亲结点；如果 index=1，返回父亲的父亲结点

返回值：如果成功返回读取的 DTDNode 的指针列表，否则返回 NULL。

5.4 索引管理(Index Manager)

5.4.1 功能介绍

OrientX 系统支持路径索引和值索引两种索引类型。用户可以为数据集的 DTD 中的某个 Element 节点或者 Attribute 节点建立单独的值索引；也可以在一个数据集上建立路径索引，

路径索引是针对数据集，也就是针对具体的某个 DTD 树的。

在 OrientX 系统中，当用户建立一个索引时，系统兵不做任何实际的操作，只是登记索引的存在。如果需要在数据集中已有的文档上建立索引，则执行将该文档插入索引的操作（参见第四部分的示例程序）；建立索引以后，当新的文档加入数据集时，它们被系统自动的加入索引中。

值得注意的是，路径索引中的索引项是按照节点的编码排序的，这样是出于能够方便的执行 Structural Join 操作。

5.4.2 路径索引接口

1. 将文档插入到索引中，提供两种参数形式，一种以文档树作为参数，另一种以数据集内的文档编号作为参数。成功返回 0，否则返回小于零的错误代码。
   int InsertDoc2PathIndex(int setID, RecordTree* recTree);
   int InsertDoc2PathIndex(int setID,int docID);
2. 在数据集上建立路径索引，以数据集名或者数聚集号作为参数，返回 0。
   int CreatePathIndex(char* DataSetName);
   int CreatePathIndex(int DataSetId);
3. 删除路径索引，以数据集名或者数聚集号作为参数，返回 0。
   int DeletePathIndex(char* DataSetName);
   int DeletePathIndex(int DataSetId);
4. 打开数据集上的路径索引，以数据集名或者数聚集号作为参数，返回 0。
   int OpenPathIndex(char* DataSetName) ;
   int OpenPathIndex(int DataSetId);
5. 关闭路径索引，因为路径索引管理器总是控制一个活跃的路径索引，索引没有参数，返回0。
   int ClosePathIndex() ;
6. 向某个路径索引项中插入一个索引节点，返回 0。
   int InsertIndexRecord(DTDNode* node,PIndexRecord record);
7. 获得某个路径索引项（一个节点列表），返回eleEid对应的节点列表。
   SearchResult* GetElementList(DTDNodeCode eleEid);
8. 基于路径索引的 Structural Join 的实现接口，返回连接的结果。其中一个是父-子连接，一个是祖先-后代连接。
   SearchResult* ParentChild(DTDNodeCode parentEid,DTDNodeCode childEid);
   SearchResult* AncestorDescendant(DTDNodeCode ancEid,DTDNodeCode desEid);

5.4.3 值索引接口

1. 在数据集的某个DTD节点上创建值索引，需要节点的类型作为参数。如果索引创建成功，返回true。
   bool Create(int setId,DTDNodeCode aid,char type);
2. 删除数据集上所有的值索引，如果成功，返回true。
   bool DropSetValueIndex(int setId);
3. 删除文档上的所有的值索引，如果成功，返回true。

bool DropDocValueIndex(int setId,int docId);

4. 删除某个值索引，如果成功，返回true。
   bool Drop(int setId,DTDNodeCode aid);
5. 打开某个值索引，如果成功，返回true。
   bool Open(int setId,DTDNodeCode aid);
6. 关闭索引，同路径索引管理器，值索引管理器只控制一个活跃的值索引。如果成功，返回true。
   bool Close();
7. 向值索引中插入一个索引项，需要它的oid和docId信息，这是值索引管理器的方法。如果插入成功，返回true。
   bool Insert(void* key,unsigned int oid,unsigned int docId);
8. 删除某个索引项，参数同 Insert 接口。如果成功，返回true。
   bool Delete(void* key,unsigned int oid,unsigned int docId);
9. 删除某个索引项，参数仅有地址信息；如果成功，返回true。
   bool Delete(void* key);
10. 检索某个索引项，返回结果放入一个 SearchResult 结构中。
    SearchResult* Search(void* key);
11. 区域检索，要求 startKey 和 endKey 满足索引中的前后顺序关系（如startKey < endKey）；返回在规定区域内的索引项。
    SearchResult* SearchRange(void* startKey,void* endKey);
12. 返回索引中的最大/最小索引项。
    void* MaxKey();
    void* MinKey();

5.5 查询执行(Query Process)

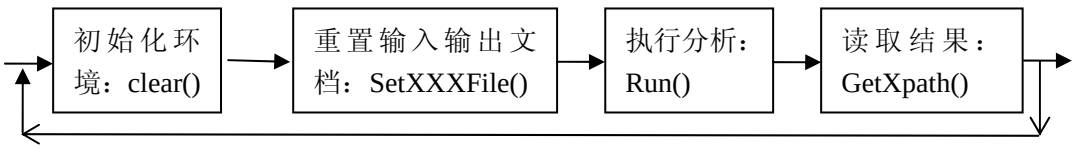
5.5.1 查询分析

查询分析的主要功能是，分析用户提交的 Xpath 查询语句；生成 Xpath 执行器所能识别的执行计划。那么它需要为上层的用户提供提交查询语句的接口；需要为下层执行器提供接口，以读取查询分析结果－－Xpath 执行计划。

查询分析模块为用户查询提供的接口是通过类 XPath_Parser 来实现的；使用 XPath_Parser 类，需要包含文件：

#include "include/QueryParser/Xpath_parser.h"

在使用XPath_Paser来进行查询分析时，接口函数的调用顺序有一定的要求，如下图所示：



XPath_Parser 类提供的函数接口说明如下：

10. 重置查询文档(查询语句以文档形式输入)filename，包括路径和文件名；默认的为标准输入stdin，成功返回1，否则返回0

    int SetInputFile(const char* filename);

11. 重置查询程序的分析结果的输出(一般输出到文档)filename，包括路径和文件名；默认的为标准输出stdout。注意:这里的输出并非查询结果的输出,而是一些为了测试结果是否正确而输出的信息。成功返回1，否则返回0

    int SetOutputFile(const char* filename);

12. 重置错误信息的输出文档filename，包括路径和文件名；默认的为标准错误stderr；如果查询语句没错,则输出"正确"。成功返回1，否则返回0

    int SetErrorFile(const char* filename);

13. 设置输出查询结果的文档filename，包括路径和文件名。采用覆盖写的方法.成功返回1，否则返回0

    int SetResultFile(const char* filename);

14. 执行分析,成功返回1,否则返回0

    int Run();

15. 初始化分析器环境；正确返回0,否则返回-1

    void clear()

16. 读取分析结果。如果查询分析失败，则返回NULL。

    CxpXpathExpr* GetXpath();

关于 Xpath 查询分析的概要设计，请参照文档“Xpath 查询分析”

5.5.2 XPath 查询执行

XPath 查询执行模块的功能是对经过 Parse 的 XPath 语句进行优化和执行。它为与用户交互的查询处理的主控模块提供接口。

XPath 查询执行内部分为查询优化和查询执行，QueryProcess 类将这两个部分整合起来，为上层提供统一、简洁的调用接口。

QueryProcess 类提供的函数接口说明如下：

1. 从单个结点 currentNode 开始（currentNode 为空表明从文档的根开始），执行分析后的 XPath 语句 path，查询结果保存在 result 中。执行成功返回 0，失败返回 1。

   int EvaluateXPath
   (ElementNode* currentNode, CxpXpathExpr* path,NxdbList& result);

2. 从具有相同 Eid 的结点集合 currentNodeList 开始，执行分析后的 XPath 语句 path，查询结果保存在 result 中。执行成功返回 0，失败返回 1。

   int EvaluateXPath
   (NxdbList& currentNodeList, CxpXpathExpr* path,NxdbList& result);

3. 返回最近的 EvaluateXPath 操作中 path 语句要查询的数据集名称。

   char* GetDataSetName();

4．返回最近的 EvaluateXPath 操作中 path 语句要查询的文档名称。

char* GetFileName();

5.5.3 XQuery 处理

XQuery 查询引擎由查询分析、查询计划优化和查询执行三个小模块组成，它通过 CxqeXQueryExecute 类型的全局变量 g_XQueryExecute 向用户提供服务，用户可以调用 XQueryExecute 的执行结果，作进一步的开发。下面是 XQuery 执行引擎所提供的主要接口。

(1) 初始化XQuery执行引擎

int initialize();

(2) 提交XQuery查询，并执行之。参数inputFile用于存放输入的查询；而outputFile则是用于查询结果的输出

int run(const char* inputFile, const char* outputFile);

(3) 清除查询引擎的环境，释放资源。注意：查询计划和查询结果都将被清空。

int clear();

(4) 读取查询执行的结果

const CComputeEleNode* getResultData()const;

CComputeEleNode* getResultData();

XPath是XQuery的一个子集，理所当然地XQuery引擎可以处理XPath。由于历史原因，XPath曾经被单独作为一个模块进行开发，它的处理策略与现在XQuery引擎又很大不一样，且所支持的功能比较少。为了能够利用原因XPath处理模块，把XQuery查询中的XPath语句片段传送给XPath查询引擎，XQuery引擎在“Include\XQueryExecute\NavigationProcessing \xqePath.h”文件中定义了宏**XPATH_MODULE**。只要定义了该宏，XQuery引擎将会把查询中的XPath片段传送到XPath处理模块中。

查询引擎实现XQuery导航式处理，有两套实现方式：一是流水线pipeline，二是物化操作materialize。通过文件“\Include\PublicHeaders\XQExeNavigation.h”中的宏 XQueryMaterialize来选择流水线操作还是物化操作。

5.5.4 更新语言

OrientX2.0 系统使用的更新语句基于对 XQuery 的扩展，用 BNF 文法表示是：

UpdateExprs ::= ForClause WhereClause UpdateClauses
|ForClause UpdateClauses

UpdateClauses ::= UpdateClauses UpdateClause
|UpdateClause

UpdateClause ::= “insert” Direction PathExpr “element” “values” Expr
|“insert” PathExpr “attribute” StringLiteral “values” Expr
| “insert” PathExpr “text” “values” Expr

```
                    |"delete"   PathExpr
                    |"delete"   PathExpr   "attribute"   StringLiteral

                    |"delete"   PathExpr   "text"

                    |"update"   PathExpr   "element" "values"   Expr
                    |"update"   PathExpr    "text"   "values"   Expr

                    |"update"   PathExpr   "attribute"   StringLiteral   "values" Expr

Direction           ::=   "before"
                    |"into"
                    |"after"
```

说明：

1) 从总体上来说可以分成两部分，一部分是 fw(for -where)语句，通过 XQuery 执行引擎定位要更新的元素；另一部分是更新子句，表述 insert、delete、update 的功能
2) ForClause、WhereClause、PathExpr、Expr 是 w3c 推荐的 XQuery 中的语法单位
3) 每个更新语句中的 PathExpr，用于定位待更新的 Element；而 Expr 则是更新值。

在插入元素时要指名插入的元素相对于参考元素的方向，before、after分别表示新元素将插入为参考元素的左兄弟和右兄弟，into表示插入为参考元素的子节点

6. 其他

关于 OrientX 系统的文件列表，类列表，类继承关系，类成员和文件索引，请参照 OrientX 系统的联机帮助文档。

如果您在使用 OrientX 系统过程中，发现系统漏洞，欢迎反馈：OrientX@ruc.edu.cn 我们将尽快给您回复解答。

谢谢您使用我们的 OrientX 系统!